1 Construction of a compiler for CL language Part I: lexical, syntax and semantic analysis

1.1 Introduction

Our goal is to implement a compiler for the CL language, that is, the programming language designed for the compilers subject. The CL language is a considerably simple Pascal-like language. We do not give an explicit and formal definition of this language by means of its grammar and semantic meaning, since one of the goals of the course is to make students to learn how to define such parts from examples. Hence, we just give some program examples that are complete enough to infer all the correct constructions of this language.

1.2 The CL language

Essentially, CL contains `INT` and `BOOL` as basic types (with the usual meaning), and type constructors `ARRAY [`⟨`size`⟩`] OF` ⟨`type`⟩, and `STRUCT` ⟨`field_list`⟩ `ENDSTRUCT` with the usual meaning.

Expressions are constructed in the classical way, with the standard binary operators "+, -, *, /, =, >, <, AND, OR", and unary operators "-, NOT". We have also predefined access (by index and by name) operators "⟨$\texttt{exp}_1$⟩`[`⟨$\texttt{exp}_2$⟩`]''` and `''`⟨`exp`⟩`.`⟨`name`⟩`''` for arrays and structs. Finally, an expression can be a function call, too.

There are instructions for assignments, conditional alternatives, and loops (see examples). There are also read-input and write-output instructions, and procedure calls.

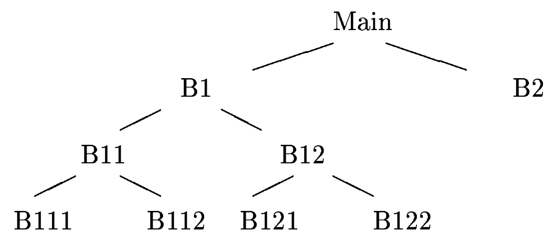
The pass mode for the parameters is either by value or by reference. See examples for the syntax of procedure and function declarations.

1.3 Scopes and visibility

CL allows nested procedures, i.e. the declaration of procedures/functions that are local/internal to other procedures/functions. Each one determines a scope.

```
Program
  Vars
    x Int
  EndVars
  Procedure B1()
    Vars
      x Int
    EndVars
    Procedure B11()
      Procedure B111()
      EndProcedure
      Procedure B112()
        x := 3
      EndProcedure
    EndProcedure
    Procedure B12()
      Procedure B121()
      EndProcedure
      Procedure B122()
        B2()
      EndProcedure
    EndProcedure
  EndProcedure
  Procedure B2()
  EndProcedure
EndProgram
```

The following is the structure of an AST for the previous program, where just the scopes have been made explicit. In fact, an alternative (and more precise) definition of scope is just the corresponding sub-AST of a procedure/function.



The visibility and scope rules for CL are the same as for Pascal:

- The corresponding definition to an occurrence of an identifier `ID` inside a scope `B` is the first one that can be found by looking at the declarations of the successive scopes that contain B, starting from B itself and so on.

In the previous example, the corresponding definition to the use of `x` in the assignment `x := 3` inside the procedure B112 is the one in the procedure B1. The procedure B121 is able to call to B121, B122, B11, B12, B1 and B2. The procedure B1 us able to call to B2, B11 and B12.

1.4 Operators priority

The priority order of the operators is depicted as follows. In case of the same priority we assume left-associativity. For example, $x-y*3+z$ is implicitely pharentized as $(x-(y*3))+z$.

Minimum priority:	AND, OR
	$=, <, >$
	$+, -$
	$*, /$
	$NOT, -$(`unary`)
Maximum priority:	$(), [], .$

1.5 The symbol table

The compiler requires to maintain a mapping from identifiers to types and other information in order to check the correctness of later expressions and instructions. This information deppends on the point of the program. The symbol table is the internal data structure of the compiler that allows to keep it. Every scope of the program contains a declaration list that will take part of this mapping when we are checking such a scope. But several scopes, with their declarations, can affect to a concrete instruction or expression. By the constraints of the previous section, a stack of informations about scopes is the adequate way to manage it, and this is exactly the implementation of the symbol table. It is just a stack of scopes (understanding now scope as information declaration of a scope). Look at the files `symtab.hh` and `symtab.cc` for the precise implementation.

1.6 Expected errors

The final avaluation of this part of the compiler is based on the correct behaviour over a set of benchmarks, after the modification of the compiler by the student in an exam. Thus, the error messages have to be exactly the ones expected. To this end, it is necessary to implement the compiler reasonably, and to make sure that it succeeds with the public benchmarks, giving exactly the expected results for them.

1.7 Plan

There is a ready-to-run compiler for a sublanguage of CL in the web page `www.lsi.upc.edu/~ggodoy/cl.html` joint to this paper. It contains several modules:

- `cl.g` (with the grammar definitions and AST construction).
- `semantic.cc` and `semantic.hh` (with the recursive function that performs the semantic analysis of the AST).

- `symtab.cc` and `symtab.hh` (with data structure definitions and functions for managing the symbol table).
- `ptype.cc` and `ptype.hh` (with data structure definitions and functions for representing and managing types).
- `compile.sh` (the script for compiling).
- `Makefile` (the makefile for compiling). Usage: "make" (built cl) or "make clean" (remove all the auxiliary files for compilation) or "make diff" (for executing with the benchmarks and comparing your results with the expected results).
- `jp1...jp12,sjp1,...,sjp12` (the benchmarks or examples of execution with expected outputs; recall that only the semantic error messages are mandatory, but not necessarily in the same order. The code generated does not need to be the same, but the result of its execution must).

The idea is to extend these modules in order to recognize any possible program of CL. In principle, it suffices to change the files `cl.g` and `semantic.cc` conveniently, but everyone should feel free to make other changes in the rest of files.

Here is a recommendable list of steps to follow in order to complete the project.

- Read the examples of the CL language, do an overview of the current modules and understand them, compile the starting project and run it with adequate examples for the basic original syntax. Extend the grammar and AST construction to the whole language. (2 sessions)
- Start the semantic analysis by incorporating the rest of basic operators in expressions. Incorporate the rest of instructions. (1 session)
- Add also the necessary checkings for array declarations and expressions. (0.5 sessions)
- Do the necessary checkings for procedure and function declarations. In our language it is possible to call to any function/procedure declared in the current scope, or in an ancestor scope of the current one. In particular, an other procedure declared later but at the same level can be called from a given procedure. Hence, names and types of functions/procedures of a scope must be inserted into the symbol table before analysing their corresponding list of instructions. This requires a first pass through the list of functions/procedures. In this pass the corresponding types are constructed and inserted in the symbol table, associated to the corresponding function/procedure identifiers. Observe that in the type construction of a function/procedure we need to look at the information of each parameter, just its type and its pass mode, but we don't need the identifier at this point. In a second pass, when we analyse the contents of every function/procedure, we stack a new scope in the symbol table, insert all the information about parameters, local variables and internal function/procedure declarations again (first pass of the next level), and check the corresponding list of instructions. The way this is structured and programmed is decision of everyone. (1 session)
- Finally, checking of procedure/function calls has to be performed. To check the list of parameters, we need both the list of real parameters (list of expressions in the AST tree) and the list of formal parameters (list of pairs $\langle$type,pass-mode$\rangle$, that is part of the type of the procedure/function already constructed and inserted in the symbol table). A simultaneous pass through such lists allows to check every real parameter with its corresponding formal parameter. (0.5 sessions)

2 Construction of a compiler for CL language Part II: code generation

2.1 Introduction

We want to construct a code translator from the CL language into a three-addresses intermediate code called t-code (t for temporal). The syntax and semantics of t-code are defined in the next subsection. However, one can skip it and look directly to the examples of translation that come joint to this paper (jp20, jp21 and so on, with their corresponding expected outputs sjp20, sjp21, etc, which also include execution results). They are intuitive and complete enough to get a good comprehension of how t-code is and how it works.

The unique goal is to make sure that our compiler generates a t-code that works the same as the one of the expected output, but they do not need to be exactly equal.

2.2 Definition of t-code

2.2.1 Syntax

A partial grammar of t-code is given as follows:

```
program: ''program'' subroutine_contents ''endprogram''
        (''subroutine'' IDENT subroutine_contents ''endsubroutine'')*;
subroutine_contents: parameters variables instructions;
parameters (IDENT)*;
variables (IDENT INTCONST)*;
instructions: (instruction)*;
instruction: INSTNAME (IDENT|TEMPORAL|EXTENDEDINTCONST|STRING)*
```

`TEMPORAL` is the set `t[0-9]+`, `IDENT` is the set `[a-zA-Z_][a-zA-Z0-9_]*`, minus the words in `TEMPORAL` and any other keyword occurring in the grammar. `INTCONST` is `[0-9]+`, and `EXTENDEDINTCONST` is `INTCONST` plus the set `offset[(]{IDENT}:{IDENT}[)]`. `STRING` is the set `["]~["]*["]`.

As you can see, every instruction has an instruction name followed by several operands. In figure 2.2.1 we show which are the allowed instruction names and how many operands and of which kind are allowed for each of them. It also contains the action of every instruction, but we will go on this later, when explaining the semantics. With respect to the allowed kind of operands, they are described as sequences of letters contained in {`a`,`s`, `i`,`t`,`v`,`c`}. An `a` means `TEMPORAL` or `IDENT`, `s` means `STRING`, `i` means `TEMPORAL` or `IDENT` or `EXTENDEDINTCONST`, `t` means `TEMPORAL`, `v` means `IDENT`, and `c` means `EXTENDEDINTCONST`.

2.2.2 Semantics

We do not want to be completelly precise when defining the memory and code management for the execution of a t-code, in order to allow different implementations (if you want a completelly formal definition you can look at the concrete implementation of an interpreter for t-code located in the file `codegest.cc`). We Assume the existence of a "big" vector of integers (4 bytes) called *memory*. The activation blocks for the different subroutine and main program activations are located in this memory. This is done in such a way that they are pushed in disjoint positions.

We also assume the existence of a mapping called *relativeoffset* giving an integer, that represents a memory index/address offset, for every subroutine or main program identifier and every identifier that is either a parameter or a local variable of such subroutine. The integer value relativeoffset[subroutine_name][variable_name] is the difference of memory positions between the address of variable_name and the address of the last parameter of subroutine_name in a given concrete activation of this subroutine, but multiplied by 4 (it is the difference in bytes, not in integer positions). The tokens `EXTENDEDINTCONST` of the form offset(ident_1:ident_2) are automatically interpreted as the integer relativeoffset[ident_1][ident_2]. This mapping is supposed to be such that the offset of every variable allows to keep as many bytes as the size related to the variable (or 4 bytes if it is a parameter), disjointly with other variables. The main program name is assumed to be "program". Thus, other subroutines must not have this name.

Figure 2.2.1: Instruction set of t-code:

Instruction name	Operand's kind	Action
addi	iia	valref(op_3):=value(op_1)+value(op_2)
aload	aa	valref(op_2):=memaddress(op_1)
call	a	calls subroutine subname(op_1)
copy	aai	moves value(op_3) bytes from memaddress(op_1) to memaddress(op_2)
divi	iia	valref(op_3):=value(op_1)/value(op_2)
equi	iia	valref(op_3):=value(op_1)=value(op_2)
etiq	v	
fjmp	ia	if not value(op_1) then currentinstruction:= label[subroutinename][labname(op_2)]
free	a	disallocates address value(op_1)
grti	iia	valref(op_3):=value(op_1)>value(op_2)
iload	ca	valref(op_2):=value(op_1)
killparam		removes a parameter
land	iia	valref(op_3):=value(op_1) and value(op_2)
lesi	iia	valref(op_3):=value(op_1)<value(op_2)
lnot	ia	valref(op_2):=not value(op_1)
load	at	valref(op_2):=memval(op_1)
loor	iia	valref(op_3):=value(op_1) or value(op_2)
mini	ia	valref(op_2):= - value(op_1)
move	vt	valref(op_2):=value(op_1)
muli	iia	valref(op_3):=value(op_1)*value(op_2)
new	ia	allocates value(op_1) bytes into valref(op_2)
popparam	a	pops a parameter into valref(op_1)
pushparam	i	pushes a parameter to value(op_1)
reai	a	reads an integer from stdin into valref(op_1)
retu		returns from the current subroutine
stop		stops the execution
stor	vv	memval(op_2):=value(op_1)
subi	iia	valref(op_3):=value(op_1)-value(op_2)
ujmp	a	currentinstruction:= label[subroutinename][labname(op_1)]
wrii	i	writes value(op_1) in stdout
wris	s	writes the string op_1 in stdout
wrln		writes an end of line in stdout

For every activation of a subroutine or main program we assume the existence of infinite temporal values t_0,t_1,t_2,..., and a variable mapping called *tempval* giving an integer for every temporal identifier.

The execution of a t-code consists on the activation of the main program, that is, the allocation in memory for its activation block, plus the execution of its list of instructions. The execution of a list of instructions can force the activation of other subroutines. The instructions call, retu, stop, pushparam, popparam and killparam serve for the management of activation and disactivation of subroutines. As you can see in figure 2.2.1, these instructions are not well precised, in order to allow different implementations of t-code executions. We assume also the existence of a memory index called *scopebase* that points to the last parameter of the currently executed subroutine, but multiplied by 4 (it is the address counting bytes instead of integer positions). The execution of a list of instructions is performed using an index that we call *currentinstruction* that initially is set to the first one. At every step, the instruction pointed by currentinstruction is executed according to the actions of figure 2.2.1. After that, currentinstruction is auto-incremented by one, and so on.

We assume a naturally computed mapping label[subroutinename][labelname] associating such a pair to the index of the code inside this subroutine with a instruction "etiq labelname". Moreover, we have a mapping labeltoint[subroutinename][labelname] giving an internal unique integer, and the coverse mapping inttolabelname[int] giving a label. Analogously, we have subroutinetoint[subroutinename] and inttosubroutinename[int].

We need to define some additional functionalities for the description of the actions of every instruction. In all cases *subroutinename* corresponds to the name of the currently executed subroutine.

- The expression value(exp) is directly exp if it is an integer value (or extended integer value that we already interpret as an integer value). Otherwise, if exp is a temporal name, then value(exp) is tempval[exp]. Otherwise, if exp is a variable name, then value(exp) is memory[(scopebase+relativeoffset[subroutinename][exp])/4]. Otherwise, if exp is a label name then value(exp) is labeltoint[subroutinename][exp], and if exp is a subroutine name then value(exp) is subroutinetoint[exp].

- The expression memaddress(exp) is tempval(exp) if exp is a temporal name, and scopebase+relativeoffset[subroutinename][exp], if it is a variable name.

- The expression memval(exp) is a reference to memory[tempval[exp]/4] if exp is a temporal name, and a reference to memory[(scopebase+relativeoffset[subroutinename][exp])/4] if exp is a variable name.

- The expression valueref(exp) is a reference to tempval[exp] if exp is a temporal name, and a reference to memory[(scopebase+relativeoffset[subroutinename][exp])/4] if exp is a variable name.

- The expression labname(exp) is exp if exp is a label name, and it is inttolabelname[value(exp)], otherwise.

- The expression subname(exp) is exp if exp is a subroutine name, and it is inttosubroutinename[value(exp)], otherwise.

In the actions of the list of instructions of figure 2.2.1 op_i is assumed to be the i-th operator of the instruction.

2.3 Plan

The translation can be naturally defined recursivelly on the AST obtained in the previous phases, which also contains semantical information like the types of the respective expressions, and the declarations of local variables, parameters, procedures and functions at every scope, with their respective types, too.

There is a ready-to-run compiler for a sublanguage of CL in the web page `www.lsi.upc.edu/~ggodoy/cl.html` joint to this paper. Apart from the modules described in the previous section, it also contains:

- `codegen.cc` and `codegen.hh` (with the functions that recursivelly define the translation from the AST).

- `codegest.cc` and `codegest.hh` (with definitions and functions for declaring and managing code, for writing it into a file, but also for directly executing it).
- `toIA32.tar.gz` contains in several files the implementation of a translator from t-code into assembler code.
- `jp20...jp33,sjp20,...,sjp33` (the examples of execution with expected outputs; recall that only the execution output is mandatory: the generated code does not need to be identical but its behaviour should be the same)

The idea is to extend these modules in order to translate any possible program of CL. It suffices to change the file `codegen.cc` conveniently. The rest of files of this part must not be modified.

Here is a recommendable list of steps to follow in order to complete the project.

- Read the examples of the CL language, do an overview of the current `codegen.cc` and understand it, compile the starting project and run it with adequate examples for the basic original syntax. Add the translation for the rest of expressions, including indexed array expressions, and the rest of instructions. For the moment, foget about the case of non-referenceable expressions of non-basic type. (2 sessions)
- Do the basic translation of procedure and function declarations, plus their corresponding calls. For the moment, forget about pass by value of non-basic type expressions, and the access to other scopes using the static link. (1 session)
- Incorporate the computation of the static link, its use when the jumped scopes of an identifier is more than 0, and the management of the auxiliar space of a scope for keeping non-referenceable values. (1 session)

2.4 Usage of the compiler

If you do not change the main function located in `cl.g`, the generated compiler will have the following parameters and functionalities, which we illustrate with examples.

The following command simply forces the compilation of `jp20`, whose result is written on the standard output, including the AST appearance, the semantic errors, and the possibly generated t-code.

```
./cl <jp20
```

If we additionally want a direct execution by the interpreter included in codegest.cc, we can do the following.

```
./cl <jp20 execute
```

Or the following, if we prefer a step by step execution.

```
./cl <jp20 executestep
```

If we want to write the resulting t-code into a file we can do:

```
./cl <jp20 writefile filename
```

If we do not specify the file name, then it is chosen to be `program.t`. Such a t-code file can be translated into assember code. Recall that we already have a t-code-to-assember translator. It has been implemented using the next version of PCCTS, i.e., ANTLR 2.7.6, which is also installed into the computers of the campus that we use. You can extract and built it by: (BE CAREFUL! It contains a file called Makefile, the same name as our previous Makefile for cl, thus lets do that in another directory).

```
gunzip toIA32.tar.gz
tar xf toIA32.tar
make
```

This generates the translator, that now can be used as follows.

```
./rc filename
```

This generates `filename.s`. Finally, we can generate an executable file and execute it.

```
gcc -o filename.x filename.s
./filename.x
```

The result should be the same as the one obtained with the direct execution of the internal interpreter.